Using Microservices to Customize Multi-Tenant SaaS: From Intrusive to Non-Intrusive

Hui Song
SINTEF, Oslo, Norway
hui.song@sintef.no

Phu H. Nguyen[1]
SINTEF, Oslo, Norway
phu.nguyen@sintef.no

Franck Chauvel
SINTEF, Oslo, Norway
franck.chauvel@sintef.no

Abstract

Customization is a widely adopted practice on enterprise software applications such as Enterprise resource planning (ERP) or Customer relation management (CRM). Software vendors deploy their enterprise software product on the premises of a customer, which is then often customized for different specific needs of the customer. When enterprise applications are moving to the cloud as mutli-tenant Software-as-a-Service (SaaS), the traditional way of on-premises customization faces new challenges because a customer no longer has an exclusive control to the application. To empower businesses with specific requirements on top of the shared standard SaaS, vendors need a novel approach to support the customization on the multi-tenant SaaS. In this paper, we summarize our two approaches for customizing multi-tenant SaaS using microservices: intrusive and non-intrusive. The paper clarifies the key concepts related to the problem of multi-tenant customization, and describes a design with a reference architecture and high-level principles. We also discuss the key technical challenges and the feasible solutions to implement this architecture. Our microservice-based customization solution is promising to meet the general customization requirements, and achieves a balance between isolation, assimilation and economy of scale.

2012 ACM Subject Classification Software and its engineering → Software as a service orchestration systems; Software and its engineering → Cloud computing; Applied computing → Service-oriented architectures

Keywords and phrases Customization, Software-as-a-Service (SaaS), Microservices, Multi-tenancy, Cloud, Reference Architecture

Digital Object Identifier 10.4230/OASIcs.Microservices.2017-2019.1

Funding This work is funded by the Research Council of Norway under the grant agreement number 256594 (the Cirrus project).

Acknowledgements We want to thank our colleagues at Supper Office and Visma for the fruitful collaboration in the Cirrus project.

1 Introduction

Most companies rely on enterprise software applications to drive their daily business, such as Enterprise resource planning (ERP) or Customer relation management (CRM). Because every company is unique, a standard product application cannot fit all the requirements of any company, and therefore often needs to be customized for individual customers. In

[1] Corresponding author

Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019).
Editors: Luís Cruz-Filipe, Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, Florian Rademacher, and Sabine Sachweh; Article No. 1; pp. 1:1–1:18
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

practice, a customer can easily spend ten times the cost on customization than the licence they bought for the original application [3]. Software customization is traditionally done by re-defining work flows, developing add-in applications, or even directly modifying the source code of the standard product application. In this paper, we differentiate *customization* from *configuration*: The former involves software development work whereas the latter only involves changing some values of the predefined parameters. Advanced customers often require features that are not predictable for vendors, making customization inevitable, since configuration is limited within the features that are already implemented by the vendors.

Following the trend of cloud computing, enterprise software vendors are moving from single-tenant on-premises applications to multi-tenant (Cloud-based) Software as a Service (SaaS) [6]. Customer companies no longer buy a license from the vendor and install it in their own premises. Instead, they subscribe to an online service, which is also used by other customers, known as *tenants* of the service. The SaaS model brings new challenges to the software vendors with regard to enabling customization. It is not possible for any tenant to directly edit the source code of the same product service shared by other tenants. Software vendors must enhance the SaaS model with the ability to enable tenant-specific customization in the multi-tenant context. Under such setup, customization on multi-tenant SaaS must meet three basic requirements. These requirements have been defined together with the two software vendors who are the industrial partners in the Cirrus project.

- *Isolation*: The customization for one tenant must not affect the other tenants. Tenant isolation, especially in terms of security is of paramount importance.
- *Assimilation*: Customization should not harm the performance and user experience of the SaaS. In other words, the "look and feel" of the SaaS with customization should not change compared to the original SaaS.
- *Economy of scale*: With more customers subscribe to customize the SaaS, the average cost per customer should decrease. The SaaS business model allows to make full use of the economy of scale, as multiple tenants (customers) share the same application and database instance [1]. Enabling customization for multi-tenant SaaS should still ensure the economy of scale brought by the SaaS business model.

The state of the art and practice on enabling customization for multi-tenant SaaS may still be at an early stage discussed as follows. There are enterprise software vendors that move their products to SaaS without supporting the same level of customization capabilities as their customers used to have on their own premises. As a result, a significant number of their customers are not following to the cloud [7]. Without customization capabilities, the customers lost an important weapon for tailoring the services according to their real requirements, and for continuous business innovation. Some vendors choose to support either lightweight, in-product customization by providing customers with scripting or work flow languages [24]. In this way, the customization capability is stronger than parameter configuration but still limited by the languages. It is also not ideal in terms of isolation, as the scripts are running inside the main product. Other vendors, especially the big ones such as Salesforce, choose a heavyweight direction, transforming themselves from a product into a development platform for customers to implement their own applications [21]. In this way, the customization in terms of standalone applications does not meet the assimilation requirements, as the external applications will break the consistent user experience of the product service, and also drag down the response time. More importantly, this solution requires huge investment from vendors and strong expertise from customization developers.

Using microservices is a promising direction to customize multi-tenant SaaS because microservices architectures offer several benefits. First, microservices for customization purposes can be packaged and deployed in isolation from the main product, which is an important requirement for multi-tenant context. Moreover, independent development and

deployment of microservices ease the adoption of continuous integration and delivery, and reduce, in turn, the time to market for each service. Independence also allows engineers to choose the technology that best suits one and only one service, while other services may use different programming languages, database, etc. Each service can also be operated independently, including upgrades, scaling, etc. In this paper, we discuss our two approaches of using microservices to enable customization for multi-tenant SaaS: intrusive and non-intrusive. The intrusive approach [22, 2] prescribes that the main body of customization code runs in a separate microservice, isolated from the main service (of the main product), whilst specific parts of the customization code are sent back to the main product and dynamically compiled and executed within the execution context of the main service. While intrusive customization using microservices is technically sound, its practical adoption by industry may be hindered by the intrusive way of customization code, which would be developed by "third-parties" that cannot be trusted by the software vendor to be dynamically compiled and executed within the execution context of the main service. Thus, we have evolved our approach to become non-intrusive [14, 15]. The non-intrusive approach avoids using intrusive call-back code for customization and rather orchestrates customization using the API Gateway pattern [19]. Via API Gateway(s), the APIs of the main product and the APIs of the microservices implementing customization are exposed for tenant-specific authorized access. We have demonstrated the two proposed approaches by two experimental use cases of transforming two Microsoft's reference .Net Core web applications into customizable SaaS: MusicStore [12] and eShopOnContainers [11].

Based on these two approaches, we generalize our approaches by providing a reference architecture of customizing multi-tenant SaaS by microservices, together with the general principles to achieve the requirements of isolation, assimilation and economy of scale. Whether intrusive or non-intrusive, our work has provided the designs and experiments towards novel, cloud-native architectures for customizing multi-tenant SaaS by tenant-specific microservices. A customization for a particular tenant is running as a standalone service and dynamically registered to the product service for this tenant. At runtime, the customization microservices are triggered by the product service when the latter reaches a registered extension point. They communicate with each other via REST APIs. The customization microservices are hosted by the same vendor cloud as the product service.

This paper is a report based on the investigation, design and experiments under the collaboration among a research institute and two software vendors. The objective of this paper is to provide: 1) a sample solution in the direction of using microservices in a high abstraction level, aiming at inspiring other vendors having the same customization problem for multi-tenant SaaS; and 2) a reference for researchers interested in the problem of multi-tenant customization, with a clarification of relevant concepts and research challenges. The contributions of this paper can be summarized as follows.

- We clarify the problem of muti-tenant SaaS customization with a conceptual architecture, which defines the high-level concepts involved in the problem and the relationships between these concepts.
- We provide a reference architecture of customizing multi-tenant SaaS by microservices, together with the general principles to achieve the requirements of isolation, assimilation and economy of scale.
- We identify a set of technical challenges towards implementing this reference architecture, and propose technical solutions towards these challenges.

The remainder of this paper is organized as follows: In Section 2, we give a motivational example to demonstrate the challenges of deep customization. Then, Section 3 provides a conceptual architecture of multi-tenant SaaS. Sections 4 and 5 describe our intrusive and non-intrusive approaches for enabling the customization of multi-tenant SaaS using microservices.

We generalize our solutions and provide a reference architecture for customization using microservices in Section 6. After that, Section 7 discusses the technical details in the proposed reference architecture. Section 8 presents the related work. Finally, we give our conclusions and future work in Section 9.

2 A Motivational Example

Let us consider *MuTeShop.com* (Multi-Tenant Shops) as a made-up example that captures the requirement of customization. *MuTeShop.com* offers web-based online shopping SaaS: Customers can quickly set up their own shopping website. From the *MuTeShop.com* software vendor's point of view, each customer is a *tenant* with a separate website for their *end-users* to browse and buy goods.

MuTeShop.com has to be customizable. For example, one of their key customer/tenant, e.g., *Music.MuTeShop.com*, requires that their shopping cart includes a charity donation option. Whenever an end-user adds an album into her shopping cart, she can donate some money to a designated charity, which eventually adds-up on the total checkout price. *Music.MuTeShop.com* hires a third-party consultant to implement this customization, which involves the following changes to the standard *MuTeShop.com* product. They need to change: *(i)* the database storage to be able to record the amount of donation for each item in the shopping cart, *(ii)* the business logic to account for these donations, and *(iii)* finally the user interface for end-users to choose/see how much they donate.

As a multi-tenant SaaS, *MuTeShop.com* cannot allow the consultant to modify their product source code to implement such customization, because the same database schema and the price accounting source code are shared by multiple tenants. Modifying the code for one tenant would interfere with the service to other tenants. Instead, the product service of *MuTeShop.com* should only provide the standard features that are common for all the tenants. The customization required specifically by *Music.MuTeShop.com* should be running in an isolated way, outside of the product service. When registered to the product service, this customization should modify the behaviour of the product service as stated above if and only if the user requests are bound to this tenant.

The example illustrates the problem that many SaaS vendors face: Their services are successful for some customers only if they can do deep customization. But the vendor cannot allow the same way of deep customization as for on-premises products, because they have to keep the service multi-tenant. In summary, they need a customization solution that achieves both *isolation* and *assimilation*.

3 A Conceptual Model of Customization for Multi-Tenant SaaS

Figure 1 summarizes the main concepts related to the customization of multi-tenant SaaS. There are three different roles in the ecosystem of multi-tenant SaaS customization, i.e., the Vendors who provide the main SaaS, the Customers who subscribe to the service, and the Consultants who are hired by the customers to customize the SaaS. In practice, the three roles are not necessarily taken by different companies, e.g., a customer company may have their own IT team and developers that are competent for doing customization. A vendor company may also have its own consult team who sells their own service and does customization under the customer's request.

The Product is the software produced by the vendor. It may have multiple versions. A Product Service is a running instance of a specific product version, hosted by a Product Environment, together with the Product Data. A product environment is a specific Environment,

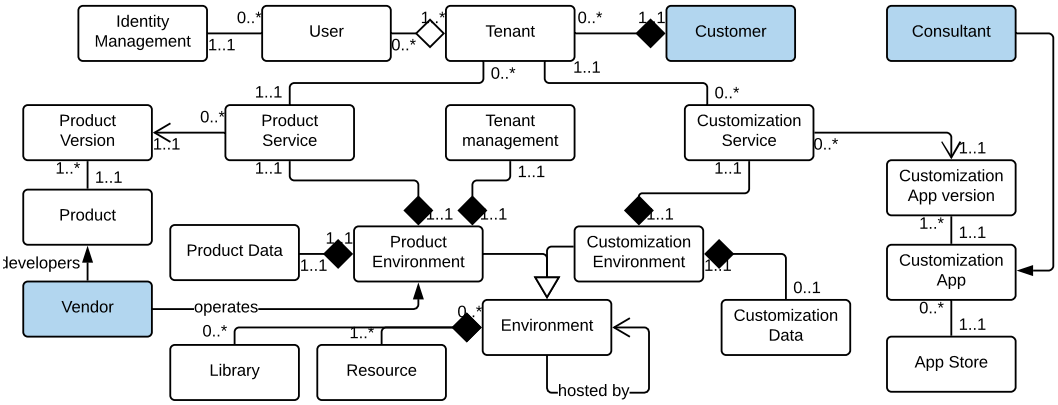

Figure 1 A conceptual model of customization for multi-tenant SaaS.

which is a self-contained computation unit with software, Libraries and Resources, such as a Docker container or a virtual machine. A product service typically runs in an exclusive environment, meaning that one environment is for one and only one instance. An environment may be hosted by another environment, e.g., a Docker container may be hosted by a virtual machine, and the latter is hosted by a cloud infrastructure.

One product service serves multiple Tenants, and at the same time, the vendor manages the product environment. A vendor usually operates multiple product services (and therefore multiple product environments) at the same time: They may need to maintain instances for several versions of the product for their customers. For the same version, they usually run one main instance for the production usage, one staging instance for user testing, and one development instance for testing and debugging. Even within the product usage, there may need several product instances due to load capacity or region constraints. Under such setup, a customer may have the subscription of multiple product instances, each of which is subscribed via a different tenant. In other words, every tenant belongs to one and only one product instance. This simplifies the billing and management for the vendors. Each tenant covers a number of Users, which are the persons who have the right to access the product instance via this tenant. They are typically employees of the customer behind this tenant, but a tenant may also include users from the consult company who help the customer through training, maintenance or customization.

A Customization App is software code that implements customization to the product. It may have several versions. A Customization Service is a running instance of one version of the Customization App. A customization service is hosted by a Customization Environment. Similar to a product environment, a customization environment also provides necessary resources, libraries and database to run the customization service. Database is optional as some lightweight customization services can be stateless. A customization service is registered to a tenant, and it only changes the behaviour of the product for this particular tenant.

4 Intrusive Customization Using Microservices

In our previous work, we have experimented with an approach to using (semi-)intrusive microservices for the customization of multi-tenant SaaS [22, 2]. We allow the tenants to replace the fine-grained structures, i.e., any part of user interface (UI), business logic (BL), or database (DB) in the original product by external microservices. The main customization

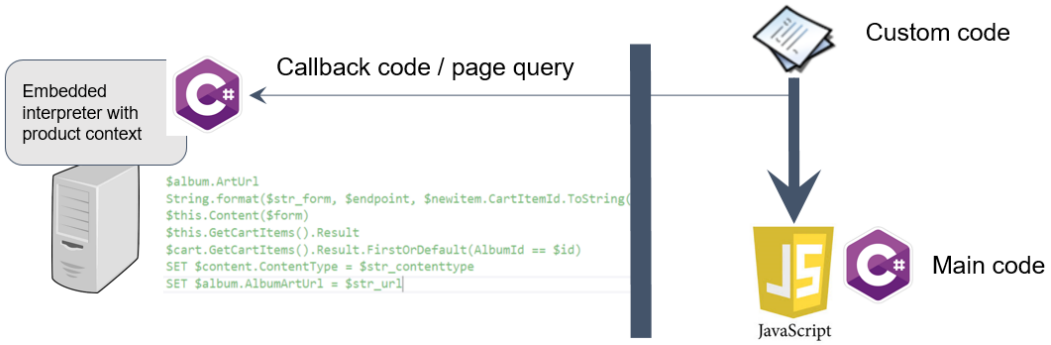

Figure 2 Intrusive customization code.

logic is running in those microservices, as parallel stacks outside of the main product. However, when a customization logic needs to access the product data or to manipulate the state of the product, it sends the so-called "call-back code" to the product, and the latter will interpret the call-back code dynamically during runtime. Since the call-back code is running under the same context as the replaced main product code, in theory it has the equivalent power as the main product code, which means that it can access any data and change any the states that are reachable by the product code. Therefore, this achieves the *deep customization* because what can be customized is not limited by the APIs of the main product. This way does not require the main product to provide any dedicated APIs for customization purpose.

A proof-of-concept implementation on multi-tenant customization based on intrusive microservices was conducted to an open source online shopping product, the Microsoft MusicStore [12]. The Microsoft MusicStore can be considered as an implementation of the MuTeShop.com example in Section 2. We first transformed it into a customizable mutli-tenant SaaS (Section 4.1). Then, we tested its customization capability by programming a simple microservice realizing the customization scenario as described in Section 2. The following proof-of-concept implementation described in Section 4.2 demonstrates the usage of synchronous triggering, intrusive invocation, separate NoSQL database, and Docker-based environments. The experiment in Section 4.2 shows that, within a reasonable cost, it is possible to enable microservice-based customization on originally un-customizable product.

4.1 Adapting the SaaS to be Customizable

We adapted the source code of MusicStore to enable microservice-based customization by adding a generic library and perform a code rewriting. A simple *tenant manager* is used to register the mapping from original methods of the main code to customization code. Figure 2 shows how the customization code interacts with the main code flow of the main product. A generic *interceptor* drives the synchronous triggering between the main product and the customization code. The *interceptor* sends the current execution context from the main product to the corresponding customization microservice for executing customization logic. After executing customization logic, the customization microservice sends callback code to the main product to apply the customization and even request for other context from the main product if follow-up customization logic is necessary. A *callback code interpreter* executes the intrusive callback code on the local context to apply the customization in the main code.

Before building and deploying the MusicStore application as a service, we performed an automatic code rewriting to enable the triggering and callback code mechanisms. In particular, we add three pieces of code in the beginning of each method. Listing 1 below shows an

example of such added code for the method *AddToCart* in the class *ShoppingCartController*. The first initializes a local context as a Dictionary object and fills it with the method parameters. This context dictionary will be used later on by the callback code interpreter. The second invokes the generic interceptor with the context and the method name. The third checks if a return value is available to decide whether to skip the original method body and return the customization results.

Listing 1 Triggering customization.

```
// first piece of code
var context = new Dictionary<string, object>()
{
    ["id"] = id,
    ["cart"] = cart,
    ["this"] = this
};
// second piece of code
ReturnValue rv = Interceptor.Intercept(
    Controllers.TenantController.currentUser,
    "MusicStore.Controllers.ShoppingCartController.AddToCart",
    false,
    context
);
// third piece of code
if(rv != null)
{
    return rv.Value;
}
```

According to our implementation practice, very light effort is required to realize the deep customization support on a legacy software application. The effort is focused on generic mechanisms, without specific consideration of the actual customization requirements or features.

4.2 Sample Customization

On the customizable MusicStore, we performed three customization use cases.

- Donation: as described in Section 2, we need to add a new page for end-users to choose the donation, a new column in the shopping cart table to show the donations and a new business logic computing total price for the shopping cart.
- Visit Counting: We want to record how many times each album has been visited. The feature needs to be triggered every time an album detail view is loaded.
- Real Cover: We want to use the album title to search the cover picture from Bing Image[2] and replace the original place-holder picture. A pop-up comment shows the picture source when the mouse cursor is hovered on the picture.

We design these use cases deliberately to achieve a good coverage of the general requirements of customization on Web-based enterprise systems. In the user interface level, they cover the changes within a web page, i.e., adding, removing or changing the position of HTML controls (UI components such as text, button, list, image, etc.), and adding a new

[2] https://www.bing.com/images/

page. The third use case also changed the browser-executed logics. In the business logic level, they cover the need to add or override server-side logics, override the action to particular events, change the bindings between UI controls and the data, and execute the services that are provided by a third party. In the database level, they require new tables, as well as new fields to an existing table. These requirements are summarized based on the actual customization cases on the on-premises products provided by the two companies.

We implemented the three customization scenarios in TypeScript, using the Node.js HTTP server to host the customization microservice [22, 2]. The first two scenarios request data storage, and we used MongoDB as the customer database. Node.js and MongoDB are running in two Docker containers, hosted on the same node as the product service. The entire customization code includes 384 lines of code in five TypeScript files (one file for each scenario, plus two common files to configure and launch the HTTP server) and 175 lines of new code in four Razor HTML templates (of which, two templates are new and the other two are copy-and-pasted from MusicStore, with 176 lines of code that are not changed).

The effect of the customized MusicStore can be seen by a screen-shot video[3]. In the video, we are using a MusicStore service through a fictional tenant. We first see the standard way to buy a music album through the MusicStore, i.e., browsing the album, add it to the shopping cart, and check the overview. After that, we deploy the customization code as a microservice and registered it into the MusicStore tenant manager. The effect of the customization is instant: When we repeat the process, we first see a new cover image of the album. When we add the album to shopping cart, we are led to a new page to select the donation amount, and shown a shopping cart overview with donations and a different total price. Finally, we open a new page to check the statistics about the album visits. At the end of the video, we log off the tenant, and the service immediately goes back to the standard behaviour.

The video also shows some non-functional features of the customization. First, the customization code is deployed and registered to the MusicStore at runtime, without rebooting the product service, and affects only the particular tenant. Second, the customized behaviour is seamlessly integrated into the product service: The new pages and the modified ones all keep the same UI style as the original MusicStore. From the end-user's perspective, it is not difficult to notice that the application has been customized.

The customization microservices have reasonable resource consumption, and is able to scale. A further examination shows that a customized page takes in average 100 millisecond longer to load, comparing to the original page. However, considering that the average page loading time in MusicStore is over two seconds, the slow down is tolerable. The footprint for a customization microservice under this scenario and the technical stack (i.e., Node.js, MongoDB, Docker) is minimal. The two Docker containers used 20 and 50 megabytes of memory at runtime.

5 Non-Intrusive Customization Using Microservices

Despite the ultimate assimilation, which means that the tenants are able to do anything for customization, just as if they are developers of the main product - our intrusive microservice approach for customization was finally not adopted by the software vendors who commissioned this research. The main concern is security. Since the partners are virtually capable of doing everything to the main product during run-time, it requires strict inspection by the vendors on the customization code, which is not pragmatic at the moment. As a result, we have turned to non-intrusive way for microservices-based customization, which should allow the vendors to keep the customization code of tenants under control [14, 15].

[3] `https://youtu.be/IIuCeTHbcxc`

To make non-intrusive customization possible, there is a main prerequisite for the web-based SaaS' architecture, i.e., the clear separation of the user interface (UI) part from the back-end business logic (BL) part [15]. This means that a web-based SaaS must be split into a WebUI part and back-end BL service(s). By separating the UI and the BL of the main product, we can introduce microservices implementing tenant-specific customization for the main product at UI, BL, and database (DB) levels. Note that we focus on web-based SaaS because of its popularity. Figure 3 shows an overview of the non-intrusive approach. Each customization for a tenant is running as a standalone microservice and dynamically registered to the main product service for this tenant. The APIs of the customization microservice are available for authorized access via API gateway. At runtime, whenever the main product service reaches a registered customization point for a tenant, the main product service triggers the corresponding customization for that tenant by calling the REST APIs of the tenant's customization microservices via the API gateway. Note that the customization microservices have been registered with the tenant manager in advance.

The WebMVC Customizer is a local component that is introduced into the main product's WebApp MVC to intercept the flows of the main product. The WebMVC Customizer is similar to the Interceptor of the intrusive approach in Section 4.1. Tenant Manager is a service that manages the customization(s) for every tenant, which is also similar to the intrusive approach. The main difference between the intrusive approach and the non-intrusive approach comes from the introduction of the API Gateways, the Identity and Access Management (IAM) service, and the Event Bus. In our non-intrusive approach, we follow the API gateway pattern [19] to decouple the client applications (e.g., the WebMVC application) from the internal microservices (for customization or main-stream BL). The key point in the non-intrusive approach is that it enables the authorized access of the tenants' customization microservices to the main product BL via the API gateways. In this way, the tenants' customization microservices can have access to the necessary execution context of the main product BL if needed and allowed. The non-intrusive approach can provide deep customization because it allows a customization service to replace a BL component of the main product for the corresponding tenant if authorized. The authorized access of the tenants' customization microservices to the main product BL components makes the deep customization manageable. This differs from the intrusive way of sending "call-back" code from customization microservices to the main product to be dynamically compiled and executed within the execution context of the main service. The IAM Service built on an OpenID Connect[4] or OAuth 2.0[5] Identity provider can make tenant-specific customization authorized. Using standardized and powerful authentication and authorization mechanisms such as OAuth 2.0 is very important for tenant-isolation at the application level, especially regarding customization. Last but not least, the Event Bus allows the customization microservices to have asynchronous event-based communication with the main product BL components for customization purposes. We have implemented a proof-of-concept of our non-intrusive approach for enabling deep customization of a reference application for microservices architecture, eShopOnContainers [11]. The MusicStore could be re-engineered to enable non-intrusive customization but we chose the eShopOnContainers because the eShopOnContainers' architecture already satisfies our prerequisites for non-intrusive customization. Due to space reason, we refer readers to [15] for more details of the proof-of-concept on eShopOnContainers.

[4] `https://openid.net/connect/`
[5] `https://oauth.net/2/`

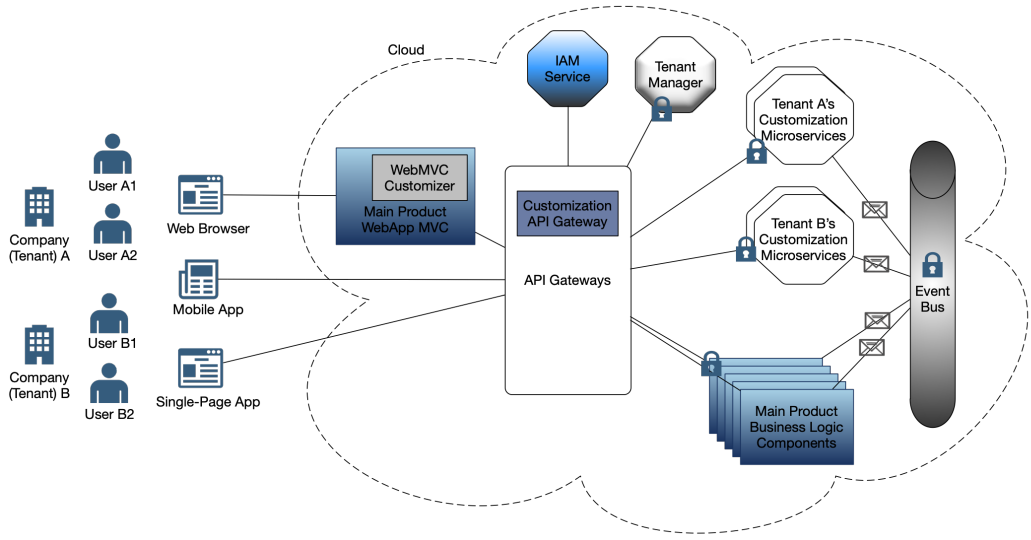

Figure 3 An overview of the non-intrusive approach [15].

6 A Reference Architecture for Customization by Microservices

This section generalizes the two customization approaches using microservices. We present the main principles for the microservice-based style for customization, and a reference architecture that follows these principles.

6.1 Principles

We adopt a microservice-based style for customization, driven by a set of high-level principles, or design decisions, in order to meet the requirements of isolation, assimilation and economy of scale. The first set of principles meets the following requirements of **isolation**.

- **Every customization is a service.** A customization should be a stand-alone running entity, with its own life-cycle independent of the product. Such a solution avoids a failure in the customization (such as dead loop) from impacting the normal operation of the main product. The interaction of customization services with the main product is monitored and administrated.
- **A customization service serves one and only one tenant.** This principle also indicates that no more than one product service connects to a same customization service, since each tenant belongs to only one product service.
- **Each customization service runs on its own environment.** This prevents customization services from influencing each other at runtime. It also simplifies the management of customization, such as monitoring and billing.
- **A customization service has its own database.** A customization should not be allowed to modify the schema of the product database. It uses its exclusive database to store the customization-specific data. A customization service does not have direct access to either the product database or the other customization database.
- **A customization service communicates with the product service and other customization services only via REST API**. Other ways of communication, such as shared database, shared memory or files, will make the services more tightly coupled.

These principles that favour isolation have negative effects on either assimilation or the economy of scale. The more isolation, the less likely that customization can be assimilated with the main product. Similarly, the more isolation leads to more use of resources, which means less favor for the economy of scale. We make the following design decisions to reduce such negative effects and bring a better balance regarding isolation, assimilation and economy of scale.

- **A customization environment is "close to" its product environment.** If a product environment is hosted by a cloud provider, such as Amazon AWS, the customization environments related to it should be deployed into an infrastructure from the same cloud provider, in the same region. This ensures the low latency of their communication.
- **The external URL to access a customization service is consistent with the product URLs.** Most of the customization services are not visible to end users, but only used indirectly when the users invoke the product. When a customization service is exposed directly to the end user, the URL to access this service should be in the same style as the product URL, so that the users do not feel the separation.
- **The vendor should provide a unified identity management for both product and customization,** so that users do not need separate login to use the customization services.
- **Customization environments should have minimal footprint**, so that the vendors can host a large scale of customers for each product service. In other words, customization services should be lightweight microservices that have minimal resource consumption.
- **The vendor should manage all the environments in an elastic way.** This will reduce the total cost of resources, especially when there is a large number of lightweight, infrequently used customization services.
- **The vendor should facilitate the reuse-by-code for customization Apps.** When a customization solution fits multiple customers (this normally happens when the customers hire the same consultant), it should be easy for the customers to reuse the customization App by code, i.e., to create a new instance of this App.

6.2 A Reference Architecture

Following the principles in the last section, we come up with a reference architecture to support the customization of multi-tenant SaaS using customer microservices. Figure 4 illustrates this reference architecture with one product instance and two customization services. The product is a typical browser-server web-based application. The single product instance serves several tenants simultaneously. The tenant management component, as part of the product instance, controls the valid tenant served by this instance, the unified identity management service controls which users have the access via each tenant. The product instance and the database are deployed in the same virtual machine from a public cloud provider. All the user requests from the browser go through a web proxy, which translate the user-friendly URL into the internal address used by the cloud provider.

Customization service customizes the behaviour of the product instance for one of its tenants, by introducing new features and replacing existing features. The customization service is registered to the tenant manager in a customization registration process before it can be triggered. The new features can be accessed via a specific URL directly from the users, or *triggered* by the product instance under predefined circumstances. The replacing features are triggered by the product instance, when the original feature in the product is about to be activated. The tenant manager defines when to trigger the customization services,

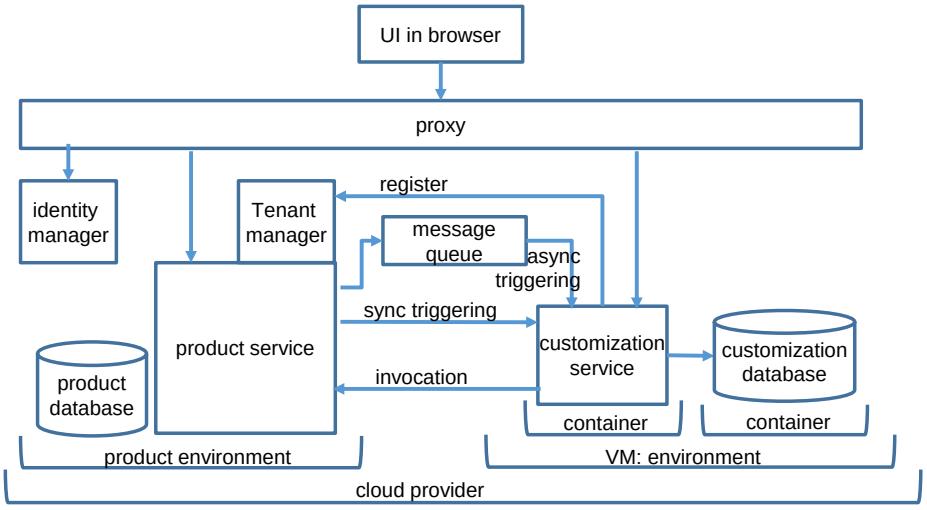

Figure 4 A reference architecture of using microservices for customization.

based on the customization registration. The customization service may need the standard data from the product instance, or modify the state and data of the product instance. It achieves this by *invoking* the product instance via API calls (in a non-intrusive approach) or call-back code (in an intrusive approach). Triggering and invoking are the two directions of communications between the product instance and the customization service. We will discuss later in Section 7 on how to implement these communications.

The customization service uses its own database to store the data that it cannot save to the standard database. The customization service and the database are deployed in two separate environments, which are in turn hosted by a virtual machine from the same provider as the product environment.

7 Discussions

This section discusses the technical challenges and potential solutions to implement the reference architecture.

7.1 Customization of Database

Customization often needs to extend the standard data type. Two types of extension on the data schema must be supported, i.e., adding a new data entity and adding a field to an existing entity. Removing an entity or a field is not necessary for customization, since the customization service can simply ignore them. Changing the type of a field can be achieved by adding a new field and ignoring the original one. Since the customization service is not allowed to change the data schema of the product database, all data under the extended entity of field has to be stored in the customization database. A new data entity can be implemented as a table in the customization database. A new field can be also implemented as a table in customization database, as a mapping from the primary key of the original table to the extended field.

The customization service registers to the tenant manager how it extends the standard data schema. In this way, the product service knows how each tenant extends its database, so that it can utilize the extended data. For example, *Music.MuTeShop.com* has a page listing

all the shopping cart items, originally with price and quantity. When rendering this page, *Music.MuTeShop.com* checks with the tenant manager and gets the information that the customization extends shopping cart items with a new field of donation amount. Therefore, it adds a new column in the shopping cart information table for this field and queries the customization service to fill in this column.

Customization databases usually have simple schema and relatively small amount of data. Therefore, it is reasonable to use light-weight technologies such as PostgreSQL and MySQL. NoSQL database is also a possibility as we have experimented with MongoDB in Section 4.2.

7.2 Triggering of Customization Services

The customization service registers itself to the tenant manager, which allows it to be triggered from one of the predefined extension points in the product service (e.g., as shown in Listing 1). When the control flow reaches this extension point, the product service picks the registered customization service, and *triggers* it. There are two types of triggering, i.e., *synchronous triggering*, when the product service awaits the customization service to finish the triggered logic, and *asynchronous triggering* when it does not.

Synchronous triggering can be implemented as a direct REST invocation from the product service to the customization service. In the product service, the implementation of an extension point can be simplified as an *if-then-else* structure: *if* the product service finds a customization service registered for this point, *then* it invokes this service and continues with the returned value, *else* it executes the standard logic. The more extension points the product service has, the more customization it supports. As an extreme case, the vendor can inject an extension point before each method in the product, using Aspect-Oriented Programming [8]. Synchronous triggering applies to the customization scenarios when the behaviour of the product service has to be influenced by the customization immediately.

Asynchronous triggering can be implemented by the event technology. At an extension point, the product service ejects an event indicating that it has reached this point, together with some context information of the extension point. The event is published to a message queue. If a customization service subscribes this message queue at the right topic, it will be notified by the message queue and triggered to handle this event. The product usually has its internal event mechanism, and therefore, to support asynchronous triggering of customization service, the vendor just needs to publish part of these internal events to the public message queue. Using asynchronous triggering, the customization cannot immediately influence the behaviour of the product service because the control flow of the product service is not blocked by the customization service.

A customization service usually needs both synchronous and asynchronous triggering. Take the visit counting scenario in Section 4.2 as an example, each time an album is visited, the customization service needs to be triggered asynchronously to increase the number of visits in its database. Later on, in the overview page, the product service needs to synchronously trigger the customization service to get the numbers of visits for all the albums. This time it needs to wait for those numbers to be returned from the customization service to show them on the overview page.

7.3 Invocation from Customization Services to the Product Service

A customization service needs to *invoke* the product service, in order to obtain the standard data and to influence the state and behaviour of the product service for the relevant tenant. From a technical point of view, there are two ways of implementing the invocation from

customization service to product service, i.e., *intrusive invocation*, when the customization service injects code into the product service, and *non-intrusive invocation*, when the customization service only relies on the APIs opened by the product service.

For intrusive invocation, the customization service send a piece of code (we call it the "callback code"), to the product service. The product service compiles and executes the callback code immediately, and sends the execution results back to the customization service. The callback code is exposed to the same context as the native code of the product service, and therefore, in theory, it can read and write all the standard data and the other states of the product service, just as a piece of native code. To support intrusive invocation, the product should have a built-in interpreter that compiles and executes the callback code. Some modern dynamic programming languages support the execution of source code from plain text, e.g., the `eval` method in Python. Microsoft .Net framework also provides the Dynamic Linq techniques to compile a query into a dynamically executable delegation. If the product is implemented in a platform without such support, the vendors can choose to translate the callback code into a set of invocations to the reflection API. For the sake of security and simplicity, the callback code should be transferred as source code, in terms of plain text, instead of binary code. The product service should provide the specific REST API for injecting callback code and return the execution result.

For non-intrusive invocation, the customization service calls the REST API of product service to obtain the standard data and to manipulate the product states. In this way, the customization capability is defined and limited by the APIs exposed to the customization services. Providing a both powerful and easy-to-use API is a big challenge for the vendor. Automatic generation of such APIs based on the data schema and the product features is a promising way.

One challenge related to the invocation of product instance, regardless of intrusive or non-intrusive way, is how to keep the execution context of the product service. In the product service, every piece of code is running under a runtime context, which is the temporary and static variables accessible by this piece of code. The context contains the important information such as the current user, the recently queried and manipulated data, etc. When the customization service kicks in and replaces an original piece of code, it normally need such context information. For intrusive invocation, a natural solution is to reserve the entire context at the point when the customization service is triggered, and the product service use this context to execute the callback code. For non-intrusive invocation, the vendor should identify the useful context information and provide specific API methods to obtain and exploit such context information.

7.4 Tenant Manager and Tenant Isolation

The tenant manager is a part of the product service, which records the customization for each tenant. When a customization service is activated, it registers to the tenant manager the following information: which tenant it customizes, what extension points it listens to (together with a service endpoint for the product service to call in order to trigger the customization logic), and how it extends the product data schema. The product service queries the tenant manager for such registration every time it reaches an extension point or requires the data extensions. Due to the frequent interaction between the product service and the tenant manager, it is reasonable to implement the tenant manager as a local component within the product service, to avoid the unnecessary overload by remote invocations.

One of the key requirements in multi-tenant SaaS is tenant isolation with security and privacy, especially together with deep customization enabled. We have better addressed this requirement in the non-intrusive approach as discussed in [15]. The non-intrusive ap-

proach [15] allows a software vendor to manage all the tenants' customization microservices, in how they are authorized to customize the main product for a specific tenant, in administrating and monitoring the customization microservices at runtime. Deploying customization microservices on separate containers for different tenants and the main product is also very important for tenant isolation as discussed below.

7.5 Customization Environments

A customization environment comprises the infrastructure, technical stack and libraries that a customization service needs at runtime. Considering that each customization service should have a unique isolated environment, and a product service may serve many tenants, a vendor cloud may host a large number of customization environments at the same time. Therefore, it is important to keep a minimal footprint for each customization environment and to simplify the management of these environments.

All the customization environments should use the same type of infrastructure, which is both light-weighted and easy to manage. The container technology, in particular Docker, appears to be very suitable for these purposes. Each customization environment is isolated in a Docker container, so as the environment that hosts a customization database. The consultant provides the vendor a Dockerfile, specifying how to construct a customization environment container, i.e., choosing an operating system, installing the technical stack step by step, downloading the customization App source code, and finally defining how to initialize the technical stack with the customization App. The vendor builds the Docker image according to the Dockerfile, in the vendor cloud, and instantiates a container from the image when customization service needs to be on-board for the tenant.

The vendor should maintain a Docker cluster composing by a flexible number of virtual machines from the same cloud provider. Depending on the number of customization services and the load on them, the vendor cloud should scale in or out of the Docker cluster. The vendor can also apply a standard management tool to monitor the state and resource consumption of each container, and kill the customization services when necessary. Such scaling and management functionality can be implemented using Docker tools.

8 Related Work

Software Product Line (SPL) [18] captures the variety of user requirements in a global variability model, and actual products are generated based on the configuration of the variability model. Traditional SPL approaches target all the potential user requirements by the software vendor, and thus do not apply to our definition of customization. Dynamic SPL [5] is closer to customization, and some approaches such as [10] propose the usage of variability models for customization. However, such model-based configuration is in a much higher abstraction level than programming [20], and focused on how to combine existing features. In contrary, customization is targeting the development of new features specific to the customers.

There are many approaches to SaaS customization in the context of service-oriented computing. However, most of approaches focus on a high-level modification of the service composition. Mietzner and Leymann [13] present a customization approach based on the automatic transformation from a variability model to BPEL process. Here customization is a re-composition of services provided by vendors. Tsai and Sun [24] follow the same assumption, but propose multiple layers of compositions. All the composite services (defined by processes) are customizable until reaching atomic services, which are, again, assumed to be provided by

the vendors. Nguyen et al. [16] develop the same idea, and introduce a service container to manage the lifecycle of composite services and reduce the time to switch between tenants at runtime. These service composition approaches all support customization in a coarse-grained way, and rely on the vendors to provide adequate "atomic services" as the building blocks for customized composite services. The microservice architecture discussed in this paper is targeted at how to allow customers to develop the atomic services and integrate them into the product service.

As market leading SaaS for CRM and ERP, the Salesforce platform and the Oracle NetSuite provide built-in scripting languages [21][9][17] for fine-grained, code-level customization. Using these scripting languages, the customization is running within the product, which requires an advanced scripting interpreter to guarantee the isolation between customization and the product. The customization capability is limited by the expression power of the language, and learning these languages is a special burden for the customization developers. Implementing customization as microservices solves these problems: Developers can choose the technical stack that suits them, and still do not need to care about the hosting of these services.

Middleware techniques are also used to support the customization of SaaS. Guo et al. [4] discuss, in a high abstraction level, a middleware-based framework for the development and operation of customization, and highlight the key challenges. Walraven et al. [25] implemented such a customization enabling middleware. In particular, they allow customers to develop customization code using the same language as the main product, and use Dependency Injection to dynamically inject these customization Java classes into the main service, depending on the current tenant. Later work from the same group [26] developed this idea and focused on the challenges of performance isolation and latency of customization code switching. The dependency injection way for customization is close to our work, in terms of the assimilation between custom code and the main service. However, operating the customization code as an external microservice eases performance isolation. A misbehavior of the customization code only fails the underlying container, and the main product only perceives a network error, which will not affect other tenants. Besides, external microservices ease management: scaling independently resource-consuming customization and eventually billing tenants accurately.

This paper is a full extension of the position paper [23]. In this paper, we have summarized our two approaches in [2] and [15] and then presented the full reference architecture for customizing multi-tenant SaaS using microservices. Moreover, we have given discussions on the technical challenges and potential solutions to implement the reference architecture.

9 Conclusion and Future Work

In this paper, we have presented a customization solution for multi-tenant SaaS using microservices. From an intrusive approach, we have evolved our solution to introduce a non-intrusive approach that could be more practical for industry. Based on these two approaches, we have provided a generalized reference architecture for enabling customization of multi-tenant SaaS using microservices. Our discussions on the technical challenges and potential solutions to implement the reference architecture give more insights for readers to adopt our customization solution. Our microservice-based customization solution is promising to meet the general customization requirements, and achieves a balance between isolation, assimilation and economy of scale. Our future research will focus on the quality assurance of the customization services, including automatic testing and online monitoring, to achieve a DevOps way of continuous customization development.

References

1 Cor-Paul Bezemer and Andy Zaidman. Multi-tenant SaaS applications: maintenance dream or nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 88–92. ACM, 2010.

2 Franck Chauvel and Arnor Solberg. Using Intrusive Microservices to Enable Deep Customization of Multi-tenant SaaS. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 30–37, September 2018. doi:10.1109/QUATIC.2018.00015.

3 Denise Ganly, Andy Kyte, Nigel Rayner, and Carol Hardcastle. The Rise of the Postmodern ERP and Enterprise Applications World. Gartner Report ID: G00259076, April 2018. URL: https://www.gartner.com/doc/2633315?ref=mrktg-srch.

4 Chang Jie Guo, Wei Sun, Ying Huang, Zhi Hu Wang, and Bo Gao. A framework for native multi-tenancy application development and management. In *e-commerce Technology and the 4th IEEE International Conference on Enterprise Computing, e-commerce, and E-Services, 2007. CEC/EEE 2007. The 9th IEEE International Conference on*, pages 551–558. IEEE, 2007.

5 Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4), 2008.

6 IDG. 2018 Cloud Computing Survey, April 2018. URL: https://www.idg.com/tools-for-marketers/2018-cloud-computing-survey/.

7 Cindy Jutras. Cloud Financials: Having It Your Way, White paper from AICPA. URL: https://online.intacct.com/WebsiteAssets_wp_mintjutras_cloud_financials_your_way.html.

8 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

9 Thomas Kwok and Ajay Mohindra. Resource calculations with constraints, and placement of tenants and instances for multi-tenant SaaS applications. In *International Conference on Service-Oriented Computing*, pages 633–648. Springer, 2008.

10 Jaejoon Lee and Gerald Kotonya. Combining service-orientation with product line engineering. *IEEE software*, 27(3):35–41, 2010.

11 Microsoft. eShopOnContainers - Microservices Architecture and Containers based Reference Application. URL: https://github.com/dotnet-architecture/eShopOnContainers.

12 Microsoft. MusicStore test application that uses ASP.NET/EF Core. URL: https://github.com/aspnet/MusicStore.

13 Ralph Mietzner and Frank Leymann. Generation of BPEL customization processes for SaaS applications from variability descriptors. In *Services Computing, 2008. SCC'08. IEEE International Conference on*, volume 2, pages 359–366. IEEE, 2008.

14 Phu H. Nguyen, Hui Song, Franck Chauvel, and Erik Levin. Towards customizing multi-tenant Cloud applications using non-intrusive microservices. *The 2nd International Conference on Microservices, Dortmund*, 2019.

15 Phu H. Nguyen, Hui Song, Franck Chauvel, Roy Muller, Seref Boyar, and Erik Levin. Using Microservices for Non-intrusive Customization of Multi-tenant SaaS. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, pages 905–915, New York, NY, USA, 2019. ACM. doi:10.1145/3338906.3340452.

16 Tuan Nguyen, Alan Colman, and Jun Han. Enabling the delivery of customizable web services. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 138–145. IEEE, 2012.

17 Oracle. Applicaiton Development SuiteScript. URL: http://www.netsuite.com/portal/platform/developer/suitescript.shtml.

18 Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

19 Chris Richardson. Microservices patterns, 2018.

20 Marcus A Rothenberger and Mark Srite. An investigation of customization in ERP system implementations. *IEEE Transactions on Engineering Management*, 56(4):663–676, 2009.

21 Salesforce. Apex Developer Guide. URL: `https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/`.

22 Hui Song, Franck Chauvel, and Arnor Solberg. Deep customization of multi-tenant SaaS using intrusive microservices. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 97–100. ACM, 2018.

23 Hui Song, Phu H. Nguyen, Franck Chauvel, Jens Glattetre, and Thomas Schjerpen. Customizing Multi-Tenant SaaS by Microservices: A Reference Architecture. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 446–448, July 2019. `doi:10.1109/ICWS.2019.00081`.

24 Wei-Tek Tsai and Xin Sun. SaaS multi-tenant application customization. In *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pages 1–12, 2013.

25 Stefan Walraven, Eddy Truyen, and Wouter Joosen. A middleware layer for flexible and cost-efficient multi-tenant applications. In *Proceedings of the 12th International Middleware Conference*, pages 360–379. International Federation for Information Processing, 2011.

26 Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant software-as-a-service applications with service lines. *Journal of Systems and Software*, 91:48–62, 2014.